ETALIS: A Rule-based System for Complex Event Processing

Darko Anicic
FZI Forschungszentrum Informatik
Haid-und-Neu-Straße 10-14
76131 Karlsruhe, Germany
darko.anicic@fzi.de

Paul Fodor
Department of Computer Science
Stony Brook University
Stony Brook, NY 11794-4400, U.S.A
pfodor@cs.sunysb.edu

Roland Stühmer
FZI Forschungszentrum Informatik
Haid-und-Neu-Straße 10-14
76131 Karlsruhe, Germany
roland.stuehmer@fzi.de

Sebastian Rudolph
Institut AIFB
Universität Karlsruhe
Karlsruhe, Germany
rudolph@aifb.uni-karlsruhe.de

Nenad Stojanovic
FZI Forschungszentrum Informatik
Haid-und-Neu-Straße 10-14
76131 Karlsruhe, Germany
nenad.stojanovic@fzi.de

ABSTRACT

Complex Event Processing (CEP) is concerned with timely detection of complex events within a stream of atomic occurrences, and has useful applications in areas including financial services, mobile and sensor devices, click stream analysis etc. In this paper, we present an expressive formalism for specifying and combining complex events. For this language we provide both a clear declarative formal semantics as well as an effective event-driven execution model via a compilation strategy into Prolog. We also show how different consumption policies in CEP can be implemented in our formalism. We provide an implementation of the language, and present performance results of our running prototype which show that our approach is highly competitive to state-of-the-art systems.

1. INTRODUCTION

An *event* represents something that occurs, happens or changes the current state of affairs. For example, an event may signify a problem or an impending problem, a threshold, an opportunity, an information becoming available, a deviation etc. We distinguish between *atomic* and *complex* events. An atomic event is defined as an instantaneous occurrence of interest at a point in time. In order to describe more complex dynamic matters that involve several atomic events, formalisms have been created which allow for combining atomic into *complex events*, using different event operators and temporal relationships. The field of Complex Event Processing (CEP) has the task of processing streams of atomic events with the goal of detecting complex events according to meaningful event patterns.[1]

In order to provide real-time and valid response to changes, event processing is often intertwined with the processing of contextual (background) data. Location-based services (LBS), which are available today in many mobile devices, are a good example for real-time contextual processing. LBS integrate events triggered as mobile users move, with location-based data (required for the LBS operation). For instance, mobile subscribers who are near to advertising restaurants, cafés or a movie theaters are notified thereof. Other examples may be found in a variety of applications related to: social networking, location-based games, local search, asset tracking, navigation, etc. Essential for the provision of LBS is to effectively process *complex events* as well as to answer vast amounts of *queries* posed by applications.[2] To achieve that, service providers may maintain the user *state*. This state keeps record of the current *context* of a user, and it helps to figure out which LBS are at present available (regarding the user's location, his service account type, etc.). As users change their contextual situation (which is diagnosed by the detection of new events), their states are updated, too. Based on the user state, new complex events and numerous queries (required for the service operation) can be quickly dealt with. In practice, when large amounts of users and many events per user have to be handled, it is more efficient to keep their states updated rather than to process a large amount of (possible distributed) complex events and queries each time from scratch.

In this paper, we propose a new event processing strategy which detects complex events by maintaining the intermediate states. Every time an atomic event (relevant w.r.t. the set of monitored events) occurs, the system updates the internal state of complex events. Essentially, this internal state encodes what atomic events are still missing for the completion a certain complex event. Complex events are detected as soon as the last event required for their detection has occurred. Descriptions telling which occurrence of an event furthers the detection of complex events (including the relationships between complex events and events they consist of) are given by

[1]Apart from this task (also known as pattern matching), CEP further addresses other issues like event filtering, routing, transformation etc.

[2]Skyhook's system processes nearly 200 million geolocation queries for Apple's iPhone LBS. Source: `http://gigaom.com/2009/04/27/iphone-is-boosting-demand-for-location-based-services/`

deductive rules. Consequently, detection of complex events then amounts to an inferencing problem. In the following, we identify a number of benefits of our event processing model, realized via deductive rules: First, a rule-based formalism (like the one we present in this paper) is expressive enough and convenient to represent diverse complex event patterns. Second, a formal deductive procedure guarantees the correctness of the entire event processing. Unlike *reactive rules* (production rules and ECA rules), declarative rules are free of side-effects; the order in which rules are evaluated is irrelevant. Third, although it is outside the scope of this paper, a rule representation of complex events may further help in the verification of complex event patterns defined by a user (e.g., by discovering patterns that can never be detected due to inconsistency problems). Further on, we can also express responses on complex events (as complex actions), and reason about them in the same formalism [8]. Fourth, by maintaining the state of changes, the proposed event model is also capable of handling queries over the entire space (i.e. answering queries that span over multiple ongoing detections of complex events). Ultimately, the proposed event model allows for reasoning over events, their relationships, entire state, and possible contextual knowledge available for a particular domain (application). Reasoning in our event model can further be exploited to find ways to reach a given aim, which is a task that requires some intelligence. For example, an application or a service needs to reach a stable or known (desired) state. To achieve this, the system has to have a capability to reason about, or to asses states (in a changing environment). Another example is to just "track and trace" the state of any entity at any time (in order to be able to "sense and respond" in a proactive way).[3]

Technically, our approach is based on decomposition of complex event patterns into *intermediate patterns* (i.e. *goals*). The status of achieved goals is materialized as first class citizens of a fact base. These materialized goals show the progress toward completion of one or more complete event patterns. Such goals are automatically asserted by rules as relevant events occur. They can persist over a period of time "waiting" in order to support detection of a more complex goal or complete pattern. Important characteristics of these goals are that they are asserted only if they are used later on (to support a more complex goal or an event pattern), goals are all unique, and persist as long as they remain relevant (after that they can be deleted). Goals are asserted by rules which are executed in the backward chaining mode. The notable property of these rules is that they are event-driven. Hence, although the rules are executed backwards, overall they exhibit a forward chaining behavior.

The contribution of this paper consists in a novel event-driven approach for Complex Event Processing. We extend work in [7] by defining an expressive complex event description language with clear declarative formal semantics. Further on, we provide an execution model for the language, specifying algorithms that enable timely, event-driven detection of complex events. The work also extends [7, 6] by implementing different consumption policies in the proposed formalism concerned with the timely deletion of intermediate goals. Finally, we provide an implementation of the language, and show experimental results of our evaluation.

The paper is organized as follows. Section 2 specifies the problem that we address in this paper. In Section 3, we introduce a new language for event processing and define its syntax. Section 4 defines the declarative semantics of the language. Section 5 provides the operational semantics of the proposed event processing model describing how complex event patterns (specified in the proposed language) can be detected in a data-driven fashion. Section 6 details different consumption policies in event processing, and shows how we implement them. We provide an implementation of our formalism, and give performance results of a prototype implementation in Section 7. Section 8 reviews existing work in this area, and compares it to ours. Finally, Section 9 summarizes the paper, and give an outline of the future work.

2. PROBLEM STATEMENT

The general task of Complex Event Processing can be described as follows. Within some dynamic setting, events take place. Those *atomic events* are instantaneous, i.e., they happen at one specific point in time and have a duration of zero. Notifications about these occurred events together with their timestamps and possibly further associated data (such as involved entities, numerical parameters of the event, or provenance data) enter the CEP system in the order of their occurrence.[4]

The CEP system further features a set of *complex event descriptions*, by means of which *complex events* can be specified as temporal constellations of atomic events. The complex events thus defined can in turn be used to compose even more complex events and so forth. As opposed to atomic events, those complex events are not considered instantaneous but are endowed with a time *interval* denoting when the event started and when it ended.

The purpose of the CEP system is now to detect complex events within this input stream of atomic events. That is, the system is supposed to notify that the occurrence of a certain complex event has been detected, as soon as the system is notified of an atomic event that completes a sequence which makes up the complex event due to the complex event description. This notification may be accompanied by additional information composed from the atomic events' data. As a consequence of this detection (and depending on the associated data), responding actions can be taken, yet this is outside the scope of this paper.

In summary, the problem we address in our approach is to detect complex events (specified in an appropriate formal language) within a stream of atomic events. Thereby we assume that the timeliness of this detection is crucial and algorithmically optimize our method towards a fast response behavior.

3. SYNTAX

In this section we present the formal syntax of the *ETALIS Language for Events*, while in the remaining sections of the paper, we will gradually introduce other aspects of the language (i.e. the declarative and operational semantics as well as the performance of a prototype based on the language[5]).

The syntax of the ETALIS language allows for the description of *time* and *events*. We represent time instants as well as durations as nonnegative rational numbers $q \in \mathbb{Q}^+$. Events can be atomic or complex. An *atomic event* refers to an instantaneous occurrence of interest. Atomic events are expressed as ground atoms (i.e. predicates followed by arguments which are terms not containing variables). Intuitively, the arguments of a ground atom describing an atomic event denote information items (i.e. event data) that provide additional information about that event.

[3]These requirements have been recognized as main characteristics of goal-directed reasoning in event processing: `http://tibcoblogs.com/cep/2008/03/04/goal-directed-event-processing`

[4]The phenomenon of *out-of-order events* meaning delayed notification about events that have happened earlier, is outside the focus of this paper.

[5]Our prototype, ETALIS, is an open source project, available at: `http://code.google.com/p/etalis`

Atomic events can be composed to form *complex events* via *event patterns*. We use event patterns to describe how events can (or have to) be temporally situated to other events or absolute time points. The language P of event patterns is formally defined by

$$P ::= \mathtt{pr}(t_1, \ldots, t_n) \quad | \; P \text{ WHERE } t \mid q \mid (P).q \\ \quad\quad\quad\quad\quad\quad\quad\quad | \; P \text{ BIN } P \mid \text{NOT}(P).[P, P]$$

Thereby, $\mathtt{pr}$ a predicate name with arity n, t_i denote terms, t is a term of type boolean, q is a nonnegative rational number, and BIN is one of the binary operators SEQ, AND, PAR, OR, EQUALS, MEETS, EQUALS, STARTS, or FINISHES. As a side condition, in every expression p WHERE t, all variables occurring in t must also occur in the pattern p.

Finally, an *event rule* is defined as a formula of the shape

$$\mathtt{pr}(t_1, \ldots, t_n) \leftarrow p$$

where p is an event pattern containing all variables occurring in $\mathtt{pr}(t_1, \ldots, t_n)$.

After introducing the formal syntax of our formalism, we will give some examples to provide some intuitive understanding before proceeding with the formal semantics in the next section. Adhering to a stock market scenario, one instantaneous event (not requiring further specification) might be `market_closes()`. Other events with additional information associated via arguments would be $\mathtt{bankrupt}(lehman)$ or $\mathtt{buys}(citigroup, wachovia)$. Within patterns, variables instead of constants may occur as arguments, whence we can write $\mathtt{bankrupt}(X)$ as a pattern matching all bankruptcy events irrespective of the victim. “Artificial” time-point events can be defined by just providing the according timestamp.

Figure 1 demonstrates the various ways of constructing complex event descriptions from simpler ones in the ETALIS Language for Events. Moreover, the figure informally introduces the semantics of the language, which will further be defined in Section 4.

Let us assume that instances of three complex events, P_1, P_2, P_3, are occurring in time intervals as shown in Figure 1. Vertical dashed lines depict different time units, while the horizontal bars represent detected complex events for the given patterns. In the following, we give the intuitive meaning for all patterns from the figure:

- $(P_1).3$ detects an occurrence of P_1 if it happens within an interval of length 3.

- P_1 SEQ P_3 represents a sequence of two events, i.e. an occurrence of P_1 is followed by an occurrence of P_3; thereby P_1 must end before P_3 starts.

- P_2 AND P_3 is a pattern that is detected when instances of both P_2 and P_3 occur no matter in which order.

- P_1 PAR P_2 occurs when instances of both P_2 and P_3 happen, provided that their intervals have a non-zero overlap.

- P_2 OR P_3 is triggered for every instance of P_2 or P_3.

- P_1 DURING (0 SEQ 6) happens when an instance of P_1 occurs during an interval; in this case, the interval is built using a sequence of two atomic time-point events (one with $q = 0$ and another with $q = 6$, see the syntax above).

- P_1 EQUALS P_3 is triggered when the two events occur exactly at the same time interval.

- NOT$(P_3).[P_1, P_1]$ represents a negated pattern. It is defined by a sequence of events (delimiting events) in the square brackets where there is no occurrence of P_3 in the interval. In order to invalidate an occurrence of the pattern, an instance of P_3 must happen in the interval formed by the end time of the first delimiting event and the start time of the second delimiting event. In this example delimiting events are just two instances of the same event, i.e. P_1. Different treatments of negation are also possible, however we adopt one from [2].

- P_3 STARTS P_1 is detected when an instance of P_3 starts at the same time as an instance of P_1 but ends earlier.

- P_3 FINISHES P_2 is detected when an instance of P_3 ends at the same time as an instance of P_1 but starts later.

- P_2 MEETS P_3 happens when the interval of an occurrence of P_2 ends exactly when the interval of an occurrence of P_3 starts.

It is worth noting that the defined pattern language captures the set of all possible 13 relations on two temporal intervals as defined in [4]. The set can also be used for rich temporal reasoning.

In this example, event patterns are considered under the *unrestricted policy*. In event processing, consumption policies deal with an issue of *selecting* particular events occurrences when there are more than one event instance applicable and *consuming* events after they have been used in patterns. We will discuss different consumption policies and their implementation in ETALIS Language for Events in Section 6.

It is worthwhile to briefly review the modeling capabilities of the presented pattern language. For example, one might be interested in defining an event matching stock market working days:

$$\mathtt{workingDay}() \leftarrow \\ \quad \text{NOT}(\mathtt{marketCloses}())[\mathtt{marketOpens}(), \mathtt{marketCloses}()].$$

Moreover, we might be interested in detecting the event of two bankruptcies happening on the same market working day:

$$\mathtt{dieTogether}(X, Y) \leftarrow \\ \quad \big(\mathtt{bankrupt}(X) \text{ SEQ } \mathtt{bankrupt}(Y)\big) \text{ DURING } \mathtt{workingDay}().$$

This event rule also shows, how event information (about involved institutions, provenance, etc.) can be “passed” on to the defined complex events by using variables. Furthermore, variables may be employed to conditionally group events into complex ones if they refer to the same entity:

$$\mathtt{indirectlyAcquires}(X, Y) \leftarrow \mathtt{buys}(Z, Y) \text{ AND } \mathtt{buys}(X, Z)$$

Even more elaborate constraints can be put on the applicability of a pattern by endowing it with a boolean type term as filter.[6] Thereby, we can detect a stock prize increase of at least 50% in a time frame of 7 days.

$$\mathtt{remarkableIncrease}(X) \leftarrow \\ \quad \big(\mathtt{prize}(X, Y_1) \text{ SEQ } \mathtt{prize}(X, Y_2)\big).7 \text{ WHERE } Y_2 > Y_1 \cdot 1.5$$

This small selection arguably demonstrates the expressivity and versatility of the introduced ETALIS language.

[6]Note that also comparison operators like $=, <$ and $>$ can be seen as boolean-typed binary functions and, hence, fit well into the framework.

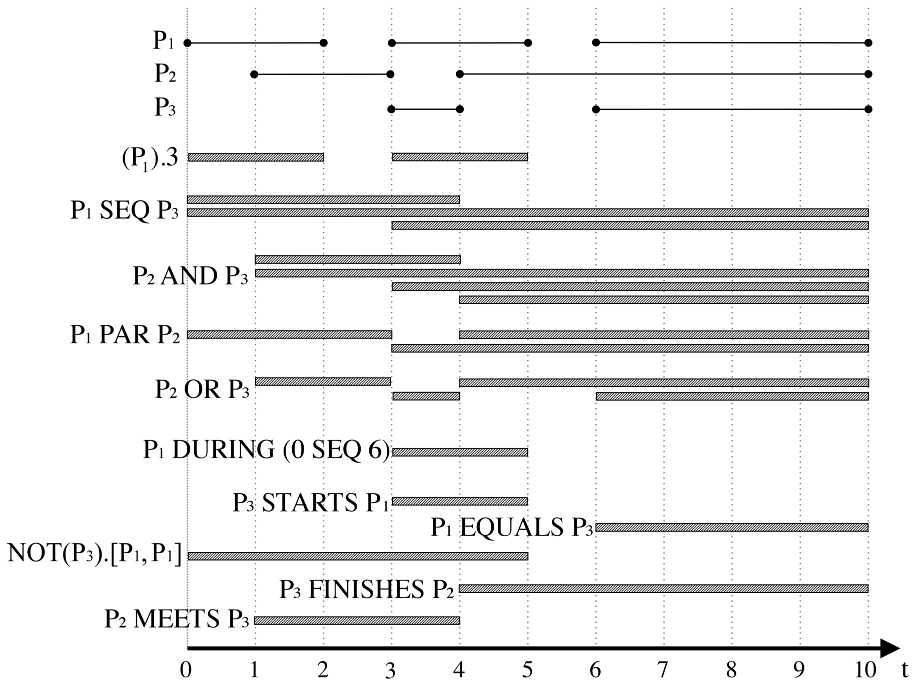

Figure 1: ETALIS Language for Events - Composition Operators

4. DECLARATIVE SEMANTICS

We define the declarative formal semantics of ETALIS Language for Events in a model-theoretic way.

Note that we assume a fixed interpretation of the occurring function symbols, i.e. for every function symbol f of arity n, we presume a predefined function $f^* : Con^n \rightarrow Con$. That is, in our setting, functions are treated as built-in utilities.

As usual, a *variable assignment* is a mapping $\mu : Var \rightarrow Con$ assigning a value to every variable. We let μ^* denote the extension of μ to terms defined in the usual way:

$$\mu^* : \begin{cases} v & \mapsto \mu(v) & \text{if } v \in Var, \\ c & \mapsto c & \text{if } c \in Con, \\ f(t_1, \ldots, t_n) & \mapsto f^*(\mu^*(t_1), \ldots, \mu^*(t_n)) & \text{otherwise.} \end{cases}$$

In addition to the set of rules $\mathcal{R}$, we fix an *event stream*. The event stream is formalized as a mapping $\epsilon : Ground \rightarrow 2^{\mathbb{Q}^+}$ from ground predicates into sets of nonnegative rational numbers. It thereby indicates at what time instants what elementary events occur. As a side condition, we require ϵ to be free of accumulation points, i.e. for every $q \in \mathbb{Q}^+$, the set $\{q' \in \mathbb{Q}^+ \mid q' < q \text{ and } q' \in \epsilon(g) \text{ for some } g \in Ground\}$ is finite.

Now, we define an interpretation $\mathcal{I} : Ground \rightarrow 2^{\mathbb{Q}^+ \times \mathbb{Q}^+}$ as a mapping from the ground atoms to sets of pairs of nonnegative rationals, such that $q_1 \leq q_2$ for every $\langle q_1, q_2 \rangle \in \mathcal{I}(g)$ for all $g \in Ground$.

Given an event stream ϵ, an interpretation $\mathcal{I}$ is called a *model* for a rule set $\mathcal{R}$ – written as $\mathcal{I} \models_\epsilon \mathcal{R}$ – if the following conditions are satisfied:

C1 $\langle q, q \rangle \in \mathcal{I}(g)$ for every $q \in \mathbb{Q}^+$ and $g \in Ground$ with $q \in \epsilon(g)$

C2 for every rule $atom \leftarrow pattern$ and every variable assignment μ we have $\mathcal{I}_\mu(atom) \subseteq \mathcal{I}_\mu(pattern)$ where $\mathcal{I}_\mu$ is inductively defined as displayed in Fig. 2.

Given an interpretation $\mathcal{I}$ and some $q \in \mathbb{Q}^+$, we let $\mathcal{I}|_q$ denote the interpretation defined via $\mathcal{I}|_q(g) = \mathcal{I}(g) \cap \{\langle q1, q2 \rangle \mid q2 - q1 \leq q\}$.

Given two interpretations $\mathcal{I}$ and $\mathcal{J}$, we say that $\mathcal{I}$ is *preferred* to $\mathcal{J}$ if there exists a $q \in \mathbb{Q}^+$ with $\mathcal{I}|_q \subset \mathcal{J}|_q$.

A model $\mathcal{I}$ is called *minimal* if there is no other model preferred to $\mathcal{I}$. It is easy to show that for every event stream ϵ and rule base $\mathcal{R}$ there is a unique minimal model $\mathcal{I}^{\epsilon,\mathcal{R}}$.

Finally, given an atom a and two rational numbers q_1, q_2, we say that the event $a^{[q_1,q_2]}$ is a *consequence* of the event stream ϵ and the rule base $\mathcal{R}$ (written $\epsilon, \mathcal{R} \models a^{[q_1,q_2]}$), if $\langle q_1, q_2 \rangle \in \mathcal{I}_\mu^{\epsilon,\mathcal{R}}(a)$ for some variable assignment μ.

It can be easily verified that the behavior of the event stream ϵ beyond the time point q_2 is irrelevant for determining whether $\epsilon, \mathcal{R} \models a^{[q_1,q_2]}$ is the case.[7] This justifies to take the perspective of ϵ being only partially known (and continuously unveiled along a time line) while the task is to detect event-consequences as soon as possible.

5. OPERATIONAL SEMANTICS

In Section 4 we have defined complex events patterns formally. This section describes how complex events, described in ETALIS Language for Events, can be detected at run-time (following the semantics of the language). Our approach is established on *goal-directed, event-driven* rules and decomposition of complex event patterns into *two-input intermediate events* (i.e. *goals*). Goals are automatically asserted by rules as relevant events occur. They can persist over a period of time "waiting" to support detection of a more complex goal. This process of asserting more and more complex goals shows the progress towards detection of a complex event. In the following subsection, we give more details about a

[7]More formally, for any two event streams ϵ_1 and ϵ_2 with $\epsilon_1(g) \cap \{\langle q, q' \rangle \mid q' \leq q_2\} = \epsilon_2(g) \cap \{\langle q, q' \rangle \mid q' \leq q_2\}$ we have that $\epsilon_1, \mathcal{R} \models a^{[q_1,q_2]}$ exactly if $\epsilon_2, \mathcal{R} \models a^{[q_1,q_2]}$.

pattern	$\mathcal{I}_\mu$(pattern)
$\mathtt{pr}(t_1, \ldots, t_n)$	$\mathcal{I}(\mathtt{pr}(\mu^*(t_1), \ldots, \mu^*(t_n)))$
p WHERE t	$\mathcal{I}_\mu(p)$ if $\mu^*(t) = \mathit{true}$ $\emptyset$ otherwise.
q	$\{\langle q, q\rangle\}$ for all $q \in \mathbb{Q}^+$
$(p).q$	$\mathcal{I}_\mu(p) \cap \{\langle q_1, q_2\rangle \mid q_2 - q_1 = q\}$
p_1 SEQ p_2	$\{\langle q_1, q_4\rangle \mid \langle q_1, q_2\rangle \in \mathcal{I}_\mu(p_1)$ and $\langle q_3, q_4\rangle \in \mathcal{I}_\mu(p_2)$ for some $q_2, q_3 \in \mathbb{Q}^+$ with $q_2 < q_3\}$
p_1 AND p_2	$\{\langle \min(q_1, q_3), \max(q_2, q_4)\rangle \mid \langle q_1, q_2\rangle \in \mathcal{I}_\mu(p_1)$ and $\langle q_3, q_4\rangle \in \mathcal{I}_\mu(p_2)$ for some $q_2, q_3 \in \mathbb{Q}^+\}$
p_1 PAR p_2	$\{\langle \min(q_1, q_3), \max(q_2, q_4)\rangle \mid \langle q_1, q_2\rangle \in \mathcal{I}_\mu(p_1)$ and $\langle q_3, q_4\rangle \in \mathcal{I}_\mu(p_2)$ for some $q_2, q_3 \in \mathbb{Q}^+$ with $\max(q_1, q_3) < \min(q_2, q_4)\}$
p_1 OR p_2	$\mathcal{I}_\mu(p_1) \cup \mathcal{I}_\mu(p_2)$
p_1 EQUALS p_2	$\mathcal{I}_\mu(p_1) \cap \mathcal{I}_\mu(p_2)$
p_1 MEETS p_2	$\{\langle q_1, q_3\rangle \mid \langle q_1, q_2\rangle \in \mathcal{I}_\mu(p_1)$ and $\langle q_2, q_3\rangle \in \mathcal{I}_\mu(p_2)$ for some $q_2 \in \mathbb{Q}^+\}$
p_1 DURING p_2	$\{\langle q_3, q_4\rangle \mid \langle q_1, q_2\rangle \in \mathcal{I}_\mu(p_1)$ and $\langle q_3, q_4\rangle \in \mathcal{I}_\mu(p_2)$ for some $q_2, q_3 \in \mathbb{Q}^+$ with $q_3 < q_1 < q_2 < q_4\}$
p_1 STARTS p_2	$\{\langle q_1, q_3\rangle \mid \langle q_1, q_2\rangle \in \mathcal{I}_\mu(p_1)$ and $\langle q_1, q_3\rangle \in \mathcal{I}_\mu(p_2)$ for some $q_2 \in \mathbb{Q}^+$ with $q_2 < q_3\}$
p_1 FINISHES p_2	$\{\langle q_1, q_3\rangle \mid \langle q_2, q_3\rangle \in \mathcal{I}_\mu(p_1)$ and $\langle q_1, q_3\rangle \in \mathcal{I}_\mu(p_2)$ for some $q_2 \in \mathbb{Q}^+$ with $q_1 < q_2\}$
NOT$(p_1).[p_2, p_3]$	$\mathcal{I}_\mu(p_2$ SEQ $p_3) \setminus \mathcal{I}_\mu(p_2$ SEQ p_1 SEQ $p_3)$

Figure 2: Definition of extensional interpretation of event patterns. We use $p_{(x)}$ for patterns, $q_{(x)}$ for rational numbers, $t_{(x)}$ for terms and pr for predicates.

goal-directed, event-driven mechanism w.r.t. event pattern operators (formally defined in Section 4).

Sequence of Events. Let us consider a sequence of events represented by rule (1) (e is detected when an event a[8] is followed by b, and followed by c). We can always represent the above pattern as $e \leftarrow ((a \text{ SEQ } b) \text{ SEQ } c)$. In general, rules (2) represent two equivalent rules.[9]

$$e \leftarrow a \text{ SEQ } b \text{ SEQ } c. \tag{1}$$

$$\begin{array}{l} e \leftarrow p_1 \text{ BIN } p_2 \text{ BIN } p_3 \ldots \text{ BIN } p_n. \\ e \leftarrow (((p_1 \text{ BIN } p_2) \text{ BIN } p_3) \ldots \text{ BIN } p_n). \end{array} \tag{2}$$

We refer to this kind of "events coupling" as *binarization* of events. Effectively, in binarization we introduce *two-input* intermediate events (goals). For example, now we can rewrite rule (1) as $ie_1 \leftarrow a \text{ SEQ } b$, and the $e \leftarrow ie_1 \text{ SEQ } c$. Every monitored event (either atomic or complex), including intermediate events, will be assigned with one or more *logic rules*, fired whenever that event occurs. Using the binarization, it is more convenient to construct *event-driven* rules for three reasons. First, it is easier to implement an event operator when events are considered on "two by two" basis. Second, the binarization increases the possibility for *sharing* among events and intermediate events, when the granularity of intermediate patterns is reduced. Third, the binarization eases the *management* of rules. As we will see later in this section, each new use of an event (in a pattern) amounts to appending one or more rules to the existing rule set. However what is important for the management of rules, we don't need to *modify* existing rules when adding new ones[10].

In the following, we give more details about assigning rules to each monitored event. We also sketch an algorithm (using Prolog syntax) for detecting a sequence of events.

Algorithm 5.1 accepts as input a rule referring to a binary sequence $e_i \leftarrow a \text{ SEQ } b$, and produces event-driven backward chaining rules (i.e. executable rules) for the sequence pattern. The binarization step must precede the rule transformation. Rules, produced by Algorithm 5.1, belong to one of two different classes of rules[11]. We refer to the first class as to *goal inserting rules*. The second class corresponds to *checking rules*. For example, rule (4) belonging to the first class inserts $goal(b(\_,\_), a(T_1, T_2), e1(\_,\_)$. The rule will fire when a occurs, and the meaning of the goal it inserts is as follows: "an event a has occurred at $[T_1, T_2]$,[12] and we are waiting for b to happen in order to detect ie_1". Obviously, the goal does not carry information about times for b and ie_1, as we don't know when they will occur. In general, the *second* event in a goal always denotes the event that has just occurred. The role of the *first* event is to specify what we are waiting for to detect an event that is on the *third* position.

Algorithm 5.1 Sequence.

```
Input: event binary goal $ie_1 \leftarrow a$ SEQ $b$.
Output: event-driven backward chaining rules for SEQ operator.
Each event binary goal $ie_1 \leftarrow a$ SEQ $b$. is converted into: {
  $a(T_1, T_2) : -for\_each(a, 1, [T_1, T_2])$.
  $a(1, T_1, T_2) : -assert(goal(b(\_,\_), a(T_1, T_2), e1(\_,\_)))$.
  $b(T_3, T_4) : -for\_each(b, 1, [T_3, T_4])$.
  $b(1, T_3, T_4) : -goal(b(T_3, T_4), a(T_1, T_2), ie_1), T_2 < T_3,$
          $retract(goal(b(T_3, T_4), a(T_1, T_2), ie_1(\_,\_))), ie_1(T_1, T_4)$.
}
```

$$\begin{array}{l} for\_each(Pred, N, L) : -((FullPred = ..[Pred, N, L]), \\ event\_trigger(FullPred), (N1 is N + 1), \\ for\_each(Pred, N1, L)) \vee true. \end{array} \tag{3}$$

$$a(1, T_1, T_2) : -assert(goal(b(\_,\_), a(T_1, T_2), e1(\_,\_))). \tag{4}$$

[8]More precisely, by "an event a" is meant an *instance* of the event a.

[9]That is, if no parentheses are given, we assume all operators to be left-associative. While in some cases, like SEQ sequences, this is irrelevant, other operators such as PAR are not associative, whence the precedence matters.

[10]This holds event if patterns with negated events are added.

[11]Later on, we will introduce the rules implementing the *for each* loop.

[12]Apart from the timestamp, an event may carry other data parameters. They are omitted here for the sake of readability.

$$b(1, T_3, T_4) : -goal(b(T_3, T_4), a(T_1, T_2), ie_1), T_2 < T_3,\\ retract(goal(b(T_3, T_4), a(T_1, T_2), ie_1(\_, \_))), ie_1(T_1, T_4). \tag{5}$$

Rule (5) belongs to the second class being a *checking rule*. It checks whether certain prerequisite goals already exist in the database, in which case it triggers the more complex event. For example, rule (5) will fire whenever b occurs. The rule checks whether $goal(b(T_3, T_4), a(T_1, T_2), ie_1)$ already exists (i.e. a has previously happened), in which case the rule triggers ie_1 by calling $ie_1(T_1, T_4)$. The time occurrence of ie_1 (i.e. T_1, T_4) is defined based on the occurrence of constituting events (i.e. $a(T_1, T_2)$, and $b(T_3, T_4)$, see Section 4). Calling $ie_1(T_1, T_4)$, this event is effectively propagated either upward (if it is an intermediate event) or triggered as a finished complex event.

We see that our *backward* chaining rules compute goals in a *forward* chaining manner. The goals are crucial for computation of complex events. They show the current state of progress toward matching an event pattern. Moreover, they allow for determining the "completion state" of any complex event, at any time. For instance, we can query the current state and get information how much of a certain pattern is currently fulfilled (e.g. what is the current status of certain pattern, or notify me if the pattern is 90% completed). Further, goals can enable *reasoning* over events (e.g. answering which event occurred before some other event, although we do not know a priori what are explicit relationships between these two; correlating complex events to each other; establishing more complex constraints between them etc.). Goals can persist over a period of time. It is worth noting that *checking rules* can also delete goals. Once a goal is "consumed", it is removed from the database[13]. In this way, goals are kept persistent as long as (but not longer) than needed. In Section 6, we will return to different policies for removing goals from the database.

Finally, in Algorithm 5.1 there exist more rules than the two mentioned types (i.e. rules inserting goals and checking rules). We see that for each different event type (i.e. a and b in our case) we have one rule with a for_each predicate. It is defined by rule (3). Effectively, it implements a loop, which for any occurrence of an event goes through each rule specified for that event (predicate) and fires it. For example, when a occurs, the first rule in the set of rules from Algorithm 5.1 will fire. This first rule will then loop, invoking all other rules specified for a (those having a in the rule head). In our case, there is only one such a rule, namely rule (4). However, in general, there may be as many of these rules as usages of a particular event may be manifold in an event program (i.e. set of all event patterns). Let us observe a situation in which we want to extend our event pattern set with an additional pattern that contains the event a (i.e. additional usage of a). In this case, the rule set representing a set of event patterns needs to be updated with new rules. This can be done even at runtime. Let us assume the additional pattern to be monitored is $ie_j \leftarrow k$ SEQ a. Then the only change we need to do is to add one rule to insert a goal and one checking rule (in the existing rule set). The change is sketched as an update of Algorithm 5.1 below[14].

Updating rules from Algorithm 5.1 to accommodate an additional usage of the event a.

```
  $a(2, T_1, T_2) : -assert(goal(b(\_, \_), a(T_1, T_2), ie_1(\_, \_)))$.
  $a(3, T_1, T_2) : -goal(a(\_, \_), k(T_3, T_4), ie_j(\_, \_]), T_4 < T_1,$
  $retract(goal(a(\_, \_), k(T_3, T_4), ie_{j\_}^{(}, \_))), ie_j(T_3, T_2)$.
```

So far, we have described in detail a mechanism for event processing with *data-driven backward chaining*. We have also described the transformation of event pattern rules into rules for real-time events detection using the *sequence* operator. In general, for a given set of rules (defining complex patterns) there will be as many transformed rules as there are usages of distinct atomic events. The set of transformed rules is further accompanied by rules to implement loops (as many as there are distinct atomic events). The same procedure is repeated for intermediate events (e.g., ie_1, ie_2). The complete transformation is proportional to the number and length of user defined event pattern rules, hence such a transformation is linear, and moreover is performed at design time.

Conceptually, our backward chaining rules for the sequence operator look very similar to rules for other operators. However, in the remaining part of this section we show the algorithms for other event operators, and briefly describe them.

Conjunction of Events. Conjunction is another typical operator in event processing. An event pattern based on conjunction occurs when all events which consist that conjunction occur. Unlike the sequence operator, here the constitutive events can happen at times with no particular order between them. For example, $ie_1 \leftarrow a$ AND b defines ie_1 as conjunction of a and b.

Algorithm 5.2 Conjunction.

```
Input: event binary goal $ie_1 \leftarrow a$ AND $b$.
Output: event-driven backward chaining rules for AND operator.
Each event binary goal $ie_1 \leftarrow a$ AND $b$. is converted into: {
  $a(T_1, T_2) : -for\_each(a, 1, [T_1, T_2])$.
  $a(1, T_3, T_4) : -goal(a(\_, \_), b(T_1, T_2), ie_1(\_, \_)), retract(goal(a(\_, \_),$
  $b(T_1, T_2), ie_1(\_, \_))), T_5 = min\{T_1, T_3\}, T_6 = max\{T_2, T_4\},$
  $ie_1(T_5, T_6)$.
  $a(2, T_1, T_2) : -\neg(goal(a(\_, \_), b(T_1, T_2), ie_1(\_, \_))),$
  $assert(goal(b(\_, \_), a(T_1, T_2), ie_{1\_}^{(}, \_)))$.
  $b(T_3, T_4) : -for\_each(b, 1, [T_3, T_4])$.
  $b(1, T_3, T_4) : -goal(b(\_, \_), a(T_1, T_2), ie_1(\_, \_)), retract(goal(b(\_, \_),$
  $a(T_1, T_2), ie_1(\_, \_))), T_5 = min\{T_1, T_3\}, T_6 = max\{T_2, T_4\},$
  $ie_1(T_5, T_6)$.
  $b(2, T_1, T_2) : -\neg(goal(b(\_, \_), a(T_1, T_2), ie_1(\_, \_))), assert(goal(a(\_, \_),$
  $b(T_1, T_2), ie_1(\_, \_)))$.
}
```

Algorithm 5.2 shows the output of a transformation of *conjunction* event patterns into *data-driven backward chaining* rules (for conjunction). The procedure for dividing complex event rules into *binary event goals* is the same as in Algorithm 5.1. However, rules for *inserting* and *checking* goals are different. Both classes of rules are specific to conjunction. We have a pair of these rules created for both, an event a as well as for b. Whenever an a occurs (denoted as some interval (T_1, T_2)) the algorithm checks whether an instance of b has already happened (see rule (6) from Algorithm 5.2). An instance of b has already happened if the current database state contains $goal(a(\_, \_), b(T_1, T_2), ie_1(\_, \_))$. In this case the event $ie_1(T_5, T_6)$ is triggered (i.e. a call for $e1(T_5, T_6)$ is issued). Otherwise, a goal which states that an a has occurred is inserted (i.e. $assert(goal(b(\_, \_), a(T_1, T_2), ie_1(\_, \_)))$ is executed by rule (7)). Now if an instance of b happens later (at some $(T_3, T4)$), rule (8) will succeed (if a has previously happened). Otherwise rule (9) will insert $goal(a(\_, \_), b(T_1, T_2), ie_1(\_, \_))$.

[13]Removal of "consumed" goals is often needed for space reasons but might be omitted if events are required in a log for further processing or analyzing.

[14]Note that an *id* of rules is incremented for each next rule being added (i.e. 2,3...)

$$a(1,T_3,T_4):-goal(a(\_,\_),b(T_1,T_2),ie_1(\_,\_)),retract(goal(a(\_,\_),b(T_1,T_2),ie_1(\_,\_))),T_5=min\{T_1,T_3\},T_6=max\{T_2,T_4\},ie_1(T_5,T_6). \tag{6}$$

$$a(2,T_1,T_2):-\neg(goal(a(\_,\_),b(T_1,T_2),ie_1(\_,\_))),assert(goal(b(\_,\_),a(T_1,T_2),ie_1(\_,\_))). \tag{7}$$

$$b(1,T_3,T_4):-goal(b(\_,\_),a(T_1,T_2),ie_1(\_,\_)),retract(goal(b(\_,\_),a(T_1,T_2),ie_1(\_,\_))),T_5=min\{T_1,T_3\},T_6=max\{T_2,T_4\},ie_1(T_5,T_6). \tag{8}$$

$$b(2,T_1,T_2):-\neg(goal(b(\_,\_),a(T_1,T_2),ie_1(\_,\_))),assert(goal(a(\_,\_),b(T_1,T_2),ie_1(\_,\_))). \tag{9}$$

Concurrency. A concurrent or parallel composition of two events ($ie_1 \leftarrow a$ PAR b.) is detected when events a and b both occur, and their intervals overlap (i.e. we also say they happen *synchronously*).

Algorithm 5.3 shows what is an output of automated transformation of a *concurrent* event pattern into rules which serve a *data-driven backward chaining* event computation. The procedure for dividing complex event rules into *binary event goals* is the same (as already described), and takes place prior to the transformation. Rules for *inserting* and *checking* goals are similar to those in Algorithm 5.2. The only change in Algorithm 5.3 is a *sufficient* condition, ensuring the interval overlap (i.e. $T_3 < T_2$).

Algorithm 5.3 Concurrency.

```
Input: event binary goal $ie_1 \leftarrow a$ PAR $b$.
Output: event-driven backward chaining rules for PAR operator.
Each event binary goal $ie_1 \leftarrow a$ PAR $b$. is converted into: {
  $a(T_3,T_4):-for\_each(a,1,[T_3,T_4])$.
  $a(1,T_3,T_4):-goal(a(\_,\_),b(T_1,T_2),ie_1(\_,\_)),retract(goal(a(\_,\_),$
  $b(T_1,T_2),ie_1(\_,\_))),T_3<T_2,T_5=min\{T_1,T_3\},$
  $T_6=max\{T_2,T_4\},ie_1(T_5,T_6)$.
  $a(2,T_3,T_4):-\neg(goal(a(\_,\_),b(T_1,T_2),ie_1(\_,\_))),T_3<T_2,$
  $assert(goal(b(\_,\_),a(T_3,T_4),ie_1(\_,\_)))$.
  $b(T_3,T_4):-for\_each(b,1,[T_3,T_4])$.
  $b(1,T_3,T_4):-goal(b(\_,\_),a(T_1,T_2),ie_1(\_,\_)),retract(goal(b(\_,\_),$
  $a(T_1,T_2),ie_1(\_,\_))),T_3<T_2,T_5=min\{T_1,T_3\},$
  $T_6=max\{T_2,T_4\},ie_1(T_5,T_6)$.
  $b(2,T_3,T_4):-\neg(goal(b(\_,\_),a(T_1,T_2),ie_1(\_,\_))),T_3<T_2,$
  $assert(goal(a(\_,\_),b(T_3,T_4),ie_1(\_,\_)))$.
}
```

Disjunction. An algorithm for detecting *disjunction* (i.e. OR) of events is trivial. Disjunction operator divides rules into separate disjuncts, where each disjunct triggers the parent (complex) event. Therefore we omit presentation of the algorithm here, but later in Section 7, we present experimental results also using an implementation of this operator.

Negation. Negation in event processing is typically understood as *absence* of an event that is negated. In order to create a time interval in which we are interested to detect absence of an event, we define a negated event in scope of other complex events. Algorithm 5.4 describes how to handle negation in the scope of a sequence. It is also possible to detect negation in an arbitrarily defined time interval.

Rules for detection of negation are similar to rules from Algorithm 5.1. We need to detect a sequence (i.e. a SEQ b), and additionally to check whether an occurrence of c happened in-between the event a and b. That is why a rule $b(1,T_5,T_6)$ needs to check whether $\neg(goal(\_,c(T_3,T_4),\_))$ is true. If yes, this means that an c has not happened during a detected sequence (i.e. $a(T_1,T_2)$ SEQ $b(T_5,T_6)$), and $ie_1(T_1,T_6)$ will be triggered. It is worth noting that a non-occurrence of c is monitored from the time when an a has been detected until the beginning of an interval which the event b is detected on.

Algorithm 5.4 Negation.

```
Input: event pattern $ie_1 \leftarrow$ NOT$(c).[a,b]$.
Output: event-driven backward chaining rules for negation.
Each event binary goal $ie_1 \leftarrow$ NOT$(c).[a,b]$ is converted into: {
  $a(T_1,T_2):-for\_each(a,1,[T_1,T_2])$.
  $a(1,T_1,T_2):-assert(goal(b(\_,\_),a(T_1,T_2),e1(\_,\_)))$.
  $b(T_3,T_4):-for\_each(b,1,[T_3,T_4])$.
  $b(1,T_5,T_6):-goal(b(\_,\_),a(T_1,T_2),ie_1(\_,\_)),\neg(goal(\_,c(T_3,T_4),\_)),$
  $T_2<T_5,T_2<T_3,T_4<T_5,retract(goal(b(\_,\_),$
  $a(T_1,T_2),ie_1(\_,\_))),ie_1(T_1,T_6)))$
  $c(T_1,T_2):-for\_each(c,1,[T_1,T_2])$.
  $c(1,T_1,T_2):-assert(goal(\_,c(T_1,T_2),\_))$.
}
```

In the following part of this section we provide brief descriptions for the remaining relations between two intervals. Each relation is easily checkable with one rule.

Duration. An event happened during (i.e. DURING) another event if the interval of the first is contained in the interval of the second. Rule (10) takes two intervals as parameters. First, it checks whether all parameters are intervals using rule (11). Then it compares whether the start of the second interval (TI_2_S) is less than the start of the first interval (TI_1_S). Additionally it checks whether the end of the first interval (TI_1_E) is less than the end of the second interval (TI_2_E).

$$\begin{aligned}&duration(TI_1,TI_2):-\\&TI_1=[TI_1\_S,TI_1\_E],validTimeInterval(TI_1),\\&TI_2=[TI_2\_S,TI_2\_E],validTimeInterval(TI_2),\\&TI_2\_S@<TI_1\_S,TI_1\_E@<TI_2\_E.\end{aligned} \tag{10}$$

$$\begin{aligned}&validTimeInterval(TI):-\\&TI=[TI\_S,TI\_E],TI\_S@<TI\_E.\end{aligned} \tag{11}$$

Start Relation. We say that an event starts another if an instance of the first event starts at the same time as an instance of the second event, but ends earlier. Therefore rule (12) checks whether the start of both intervals are equal and whether the end of the first event is smaller than the end of the second one.

$$\begin{aligned}&starts(TI_1,TI_2):-\\&TI_1=[TI_1\_S,TI_1\_E],validTimeInterval(TI_1),\\&TI_2=[TI_2\_S,TI_2\_E],validTimeInterval(TI_2),\\&TI_1\_S=TI_2\_S,TI_1\_E@<TI_2\_E.\end{aligned} \tag{12}$$

Equal Relation. Two events are equal if they happen right at the same time. Rule (13) implements this relation.

$$\begin{aligned}&equals(TI_1,TI_2):-\\&TI_1=[TI_1\_S,TI_1\_E],validTimeInterval(TI_1),\\&TI_2=[TI_2\_S,TI_2\_E],validTimeInterval(TI_2),\\&TI_1\_S=TI_2\_S,TI_1\_E=TI_2\_E.\end{aligned} \tag{13}$$

Finish Relation. One event finishes another one if an occurrence of the first ends at the same time as an occurrence of the second event, but starts later. Rule (14) check this condition.

$$\begin{aligned}&finishes(TI_1,TI_2):-\\&TI_1=[TI_1\_S,TI_1\_E],validTimeInterval(TI_1),\\&TI_2=[TI_2\_S,TI_2\_E],validTimeInterval(TI_2),\\&TI_2\_S@<TI_1\_S,TI_1\_E=TI_2\_E.\end{aligned} \tag{14}$$

Meet Relation. Two events meet each other when the interval of the first ends exactly when the interval of the second event starts. Hence, the condition $TI_1\_E = TI_2\_S$ in rule (15) is sufficient to detect this relation.

$$\begin{array}{l} meets(TI_1, TI_2) : - \\ TI_1 = [TI_1\_S, TI_1\_E], validTimeInterval(TI_1), \\ TI_2 = [TI_2\_S, TI_2\_E], validTimeInterval(TI_2), \\ TI_1\_E = TI_2\_S. \end{array} \quad (15)$$

6. CONSUMPTION POLICY

When detecting a complex event, there may be several event occurrences (of the same type), that could be used to form that complex event. *Consumption policies* (or event contexts) deal with the issue of selecting particular occurrence(s), which will be used in the detection of a complex event. For example, let us consider rule (1) from Section 5, and a sequence of atomic events that happened in the following order: $a(1), a(2), a(3), b(4), b(5), c(6)$ (where an event attribute denotes a time point when an event instance has occurred.). We expect that, when an event of type b occurs, an intermediate event ie_1 must be triggered. However, the question is, which occurrence of a will be selected to build that event, $a(1)$, $a(2)$ or $a(3)$? Different consumption policies define different strategies. Here, we review three widely used consumption policies: *recent*, *chronological*, and *unrestricted* policy [12, 23], and show how they can be naturally implemented by rules in our framework.

6.1 Consumption Policy Defined on Time Points

In the above example, we assumed that the stream of events $a(1), a(2), a(3), b(4), b(5), c(6)$ contains only atomic events[15].

Recent Policy. The most recent event of its type is considered to construct complex events. In our example, when $b(4)$ occurs, $a(3)$ will be selected to compose $ie_1(3, 4)$. After a more recent occurrence $b(5)$ occurs, older (which are less recent) occurrences of b are deleted (i.e. they are no longer eligible for further compositions). The next pair, $a(3), b(5)$, is selected to form $ie_1(3, 5)$. It replaces the less recent occurrence ie_1. Finally, when $c(6)$ occurs, it will trigger $e(3, 6)$ (using $ie_1(3, 5)$ as the more recent occurrence of ie_1 in comparison to $ie_1(3, 4)$).

The recent policy can be easily implemented in our framework. Let us consider Algorithm 5.1, particularly the rule which inserts a goal (in our example, $goal(b, a, ie_1)$). Whenever an instance of a occurs, there will be a new goal inserted with a corresponding timestamp. E.g. for $a(1)$, $goal(b(\_), a(1), ie_1(\_, \_))$ is added; for $a(2)$, $goal(b(\_), a(2), ie_1(\_, \_))$, etc...). If we insert these goals into the database using LIFO (Last In First Out) structure, we obtain the *recent policy*. In our prototype implementation [5], this is done with a rule of the following form:

$$assert(goal(X)) : -asserta(goal(X)). \quad (16)$$

$asserta$ is a standard Prolog built-in that adds a term to the *beginning* of the database. Whenever a goal is inserted to the database, it is put on the top of a relation. Hence whenever we read a goal, the one inserted last will be returned.

Chronological Policy. This policy "consumes" the events in chronological order. In our example, this means that $a(1)$ and $b(4)$ will form $ie_1(1, 4)$, and further $a(2)$ followed by $b(5)$ will trigger $ie_1(2, 5)$. When $c(6)$ happens, it will trigger $e(1, 6)$.

[15]As stated before, an atomic event $e(T_1, T_2)$ is an event whose occurrence is characterized by a time point, rather than an interval, i.e. where $T_1 = T_2$.

It is straightforward to implement the chronological policy, too. Now, the goals in Algorithm 5.1 are inserted in a LIFO (Last In First Out) mode. Equivalently, we use the following rule to realize the chronological policy:

$$assert(goal(X)) : -assertz(goal(X)). \quad (17)$$

$assertz$ is a standard Prolog built-in that adds a term to the *end* of the database. Whenever a goal is inserted to the database, it is put at the end of a relation. Consequently, whenever we read a goal, the first inserted goal will be returned first.

Unrestricted Policy. In this policy, all occurrences are valid. Consequently, no event is consumed (and no event is deleted), which makes this policy not suitable for practical use. Going back to our example, this implies that we detect the following instances of ie_1: $ie_1(1, 4)$, $ie_1(2, 4)$, $ie_1(3, 4)$, $ie_1(1, 5)$, $ie_1(2, 5)$, $ie_1(3, 5)$. The event e will be triggered just as many times, that is: $e(1, 6)$, $e(2, 6)$, $e(3, 6)$...

We obtain the unrestricted policy simply by not using the construct for deleting goals (i.e. $retract$) from the database. If we replace the rule for $b(1)$ in Algorithm 5.1 with rule (18), even consumed goals will not be deleted from the database. Hence they will be available for future compositions.

$$b(1) : -goal(b, a, e1) \text{ SEQ } e1. \quad (18)$$

Consumption policies are an important part of an event processing framework. We notice that different policies change the semantics of event operators. For example, with the same operator we have detected different complex events (the recent policy detects $e(2, 6)$, while the chronological policy detects $e(1, 6)$).

6.2 Consumption Policy Defined on Time Intervals

We have so far discussed consumption policies assuming that we consider atomic events (in an input stream). As atomic events happen in time points, it is possible to establish a *total order* of their occurrences. Consequently it is easy to answer which event instance, out of two, happened more recently. When we deal with complex events ($T_1 \neq T_2$), a total order is not always possible. This subsection provides possible options in defining consumption policies in such a case.

Recent Policy. Let us consider the following sequence of input events: $a(1, 30)$, $a(15, 30)$, $b(35, 50)$. In our example rule (1) (from Section 5), the question now is which instance of a is more *recent*, $a(1, 30)$ or $a(15, 30)$? In our opinion, this question depends on the particular application domain. There are three possible options. First, an event detected on a *longer event duration* is selected to be the recent one (i.e. $a(1, 30)$). This option is suitable when aggregation functions (e.g., sum, average etc.) are applied along time windows. Hence, events detected on longer durations possibly reflect more accurate results. The second option is to choose an event with a *shorter duration* (i.e. $a(15, 30)$). This preference is e.g. suitable when indeed more recent events are desired. For example, we are interested in data (carried by events) that are as up to date as possible. Finally, the third possibility is to pick up an event instance based on *data value selection* i.e. non-temporal properties. For instance, events ending at the same time, $a(1, 30, X, Vol = 1000)$ and $a(15, 30, X, Vol = 10000)$, are selected based on an attribute value (e.g., greater $Volume$[16]).

We implement these three cases with rules (19)-(7). When an a occurs, there is a policy check performed. In rule (19), for two events with the same ending (i.e. $a(T_1, T_3)$ and $a(T_2, T_3)$) we make sure that one with a longer path ($T_1 > T_2$) is selected. In rule (20), we replace goals if the time condition is opposite ($T_1 < T_2$). Finally, in data value (or attribute value) selection, we distinguish based on a chosen attribute (e.g. $Vol_1 > Vol_2$).

$$event\_trigger(a(T_1, T_3, X, Vol_1)) : -goal(\_, a(T_2, T_3, \_, \_), \_) \text{ SEQ } T_1 > T_2 \text{ SEQ } assert(goal(\_, a(T3, T4, \_, \_), \_)) \text{ SEQ } assert(goal(\_, a(T1, T3, X, Vol_1), \_)). \tag{19}$$

$$event\_trigger(a(T_1, T_3, X, Vol_1)) : -goal(\_, a(T_2, T_3, \_, \_), \_) \text{ SEQ } T_1 < T_2 \text{ SEQ } retract(goal(\_, a(T3, T4, \_, \_), \_)) \text{ SEQ } assert(goal(\_, a(T1, T3, X, Vol_1), \_)). \tag{20}$$

$$event\_trigger(a(T_1, T_3, X, Vol_1)) : -goal(\_, a(T_2, T_3, X, Vol_2), \_) \text{ SEQ } Vol_1 > Vol_2 \text{ SEQ } retract(goal(\_, a(T3, T4, X, Vol_2), \_)) \text{ SEQ } assert(goal(\_, a(T1, T3, X, Vol_1), \_)). \tag{21}$$

Policy rules (19)-(21) are fired before inserting a new goal. It is worth noting that such an update of a goal is performed incrementally. We pay an additional price for forcing a particular consumption policy. However, the policy rules (19)-(21) are rather simple rules. In return, they ensure that no more than one goal with the same timestamp (w.r.t certain policy) is kept in memory (during processing). Hence the rules enable a better *memory management* of our framework.

Chronological Policy. The main principle in the implementation of this policy is the same as in the recent policy. The only difference is that now we consider the same start and the different ending in multiple event occurrences ($a(T1, T2)$,$a(T1, T3)$). To implement this policy, rule (19) will now contain the time condition from rule (20), and vise versa. Rule (21) remains unchanged, as well as *unrestricted policy* (which is the same as for the case with atomic events, see Subsection 6.1).

7. EXPERIMENTAL RESULTS

As a proof of concept, we have implemented the ETALIS language, in particular algorithms from Section 5. In this section, we present experimental results of tests with our prototype implementation [5]. The prototype automatically compiles the user-defined complex event descriptions into Prolog rules suitable for event-driven pattern detection. The test cases presented here were carried out on a workstation with Pentium dual-core processor 2GHz CPU and 3GB memory running on Ubuntu Linux. The ETALIS prototype was tested using each SWI Prolog version 5.6.64[17] and Yap Prolog version 5.1.3[18].

Stock Ticker Test Case. In this test scenario, we virtually monitor stocks from two companies (e.g., Google Inc. with symbol "GO", and Microsoft Corporation, "MS"). The experiment consists of six complex events (i.e. ce_1 - ce_6, see experiment rules (22) below). The first complex event, ce_1, detects every increase in price of Google stocks by more than 20%. Likewise, ce_2 detects the increase in Microsoft stocks, of the same amount. Events ce_3-ce_6 use the defined complex events ce_1 and ce_2 to build even more complex events. Event ce_3 is triggered whenever either stocks of Google or Microsoft increase by more than 20%, while ce_4 occurs when both prices increase (no matter in which order the increases happened) within a timeframe of 20 days. Event ce_5 is detected when the time spans of the two increases overlap. Finally, ce_6 occurs whenever there is a repeated increase of Google stocks (20%), with no increase of Microsoft stocks in between.

$$\begin{aligned}
&ce_1 \leftarrow (stock(\text{"}GO'', Pr1, Vol1) \text{ SEQ } stock(\text{"}GO'', Pr2, Vol2)) \\
&\qquad \text{WHERE } Pr1 < 1.20 * Pr2. \\
&ce_2 \leftarrow (stock(\text{"}MS'', Pr1, Vol1) \text{ SEQ } stock(\text{"}MS'', Pr2, Vol2)) \\
&\qquad \text{WHERE } Pr1 < 1.20 * Pr2. \\
&ce_3 \leftarrow ce_1 \text{ OR } ce_2. \\
&ce_4 \leftarrow (ce_1 \text{ AND } ce_2).20. \\
&ce_5 \leftarrow ce_1 \text{ PAR } ce_2. \\
&ce_6 \leftarrow \text{NOT}(ce_2)[ce_1, ce_1].
\end{aligned} \tag{22}$$

Event Generator 1. To test our implementation, we have implemented an event stream generator, similar to one from [3], which creates time series data. In the following experiments, we simulate stock ticker streams. All events are of type $stock(Id, Price, Volume)$, with three attributes, company *id*, *price*, and *volume*, with respective value ranges [1-2], [1-1000], [1-1000]. The price of *stock* events has probability p for increasing, $(1 - p)/2$ for decreasing, and $(1 - p)/2$ for staying the same. The value for p in our tests is 0.5. The symbol and volume follow the uniform distribution. We only consider stocks from two companies, as adding events with more than two Ids does not change the cost of processing each event (an event belongs only to one symbol). We have generated streams of different sizes, ranging from 10,000, up to 100,000 events (of previously described types).

Figure 3 shows the experimental results for the example above obtained with: Drools 5.0 Fusion[19], Esper 3.1.0[20]. and our ETALIS prototype using the SWI and Yap Prolog systems. The ETALIS prototypes are faster than both Esper and Drools, while the maximum throughputs that we got for ETALIS with input of 50,000 stock ticks were of 30,000 events/second under SWI Prolog and of 37,000 events/second under Yap Prolog. The memory consumption Figure 4, during the pattern matching process, was very low and almost constant. Irrelevant events are removed from memory as soon as they are "consumed" (i.e. propagated upward or triggered).

In this section, we have provided first measurement results. Even though there is a lot of room for improvements and optimizations, preliminary results show that logic-based event processing has the capability to handle significant amounts of events in reasonable time. Taking its strength (i.e. robust inference capability) into account, it promises a powerful approach for combining *deductive* capabilities and *temporal* features in a unified framework, while at the same time exhibiting highly competitive run-time characteristics.

8. RELATED WORK

In order to capture relevant changes in a system and respond to those changes adequately, a number of formal reactive frameworks have been proposed. Work on modeling *behavioral* aspects of an application (using various forms of reactive rules) started in the Active Database community a long time ago. Different aspects have been studied extensively, ranging from modeling and execution of rules to discussing architectural issues [21]. However, what is clearly missing in this work is a clean integration of active behavior with pure *deductive* and *temporal* capabilities.

[16] X denotes some other attribute, e.g., $Price$ etc.

[17] SWI Prolog `http://www.swi-prolog.org/`.

[18] Yap Prolog: `http://www.dcc.fc.up.pt/~vsc/Yap/`

[19] Drools Fusion: `http://www.jboss.org/drools/drools-fusion.html`

[20] Esper: `http://esper.codehaus.org/`

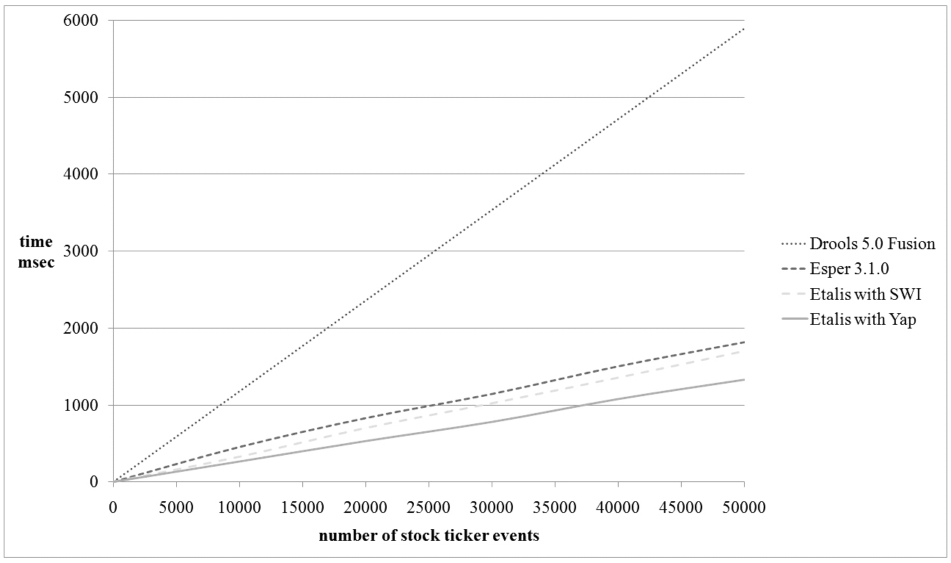

Figure 3: The event throughput for the Stock Ticker experiment

A lot of work [19, 17, 20, 10] in the area of rule-based CEP has been carried out, proposing various kinds of *logic rule*-based approaches to process complex events. As pointed out in [10], rules can be effectively used for describing so-called "virtual" event patterns. There exist a number of other reasons to use rules: Rules serve as an abstraction mechanism and offer a higher-level event description. Also, rules allow for an easy extraction of different views of the same reactive system. Rules are suitable to mediate between the same events differently represented in various interacting reactive systems. Finally, rules can be used for reasoning about causal relationships between events.

To achieve the aforementioned aims, these approaches all represent complex events as rules (or queries). Rules can then be processed either in a bottom-up manner [22], a top-down manner [13, 1], or in a manner that combines both [9]. However, all these evaluation strategies have not particularly been designed for event-driven computation. They are rather suited for a *request-response* paradigm. That is, given (and triggered by) a request, an inference engine will search for and respond with an answer. This means that, for a given event pattern, an event inference engine needs to check if this pattern has been satisfied or not. The check is performed at the time when such a request is posed. If satisfied by the time when the request is processed, a complex event will be reported. If not, the pattern is not detected until the next time the same request is processed (though it can become satisfied in-between the two checks, being undetected for the time being). For instance, [20] follows the mentioned request-response (or so called *query-driven*[21]) approach. It proposes to define queries that are processed repetitively at given intervals, e.g. every 10 seconds, trying to discover new events. However, generally events are not periodic or if so might have differing periods and nevertheless complex events should be detected as soon as they occur (not in a predefined time window). This holds in particular for time-critical scenarios such as monitoring stock markets or nuclear power plants.

To overcome this issue, in [10], an incremental evaluation was proposed. The approach is aimed at avoiding redundant computations (particularly re-computation of joins) every time a new event arrives. The authors suggest to utilize relational algebra evaluation techniques such as incremental maintenance of materialized views [15]. Still it remains unclear, how performant this solution is from [10]. Also, it is a question how different event consumption policies [11] can be handled. In particular, certain events need to be efficiently removed from certain relations (after being consumed by different complex events).

A big portion of related work in the area of rule-based CEP is grounded on the Rete algorithm [14]. Rete is an efficient pattern matching algorithm, and it has been the basis for many production rule systems (CLIPS[22], TIBCO BusinessEvents[23], Jess[24], Drools[25], BizTalk Rules Engine[26] etc.). The algorithm creates a decision tree that combines the patterns in all the rules of the knowledge base. Rete was intended to improve the speed of forward chained production rule systems at the cost of space for storing intermediate results. The left hand side of a production rule can be utilized to form a complex event pattern, in which case Rete is used for CEP. Thanks to forward chaining of rules, Rete is also event-driven (data-driven).

Close to our approach is [16]. It is an attempt to implement business rules also with a Rete-like algorithm. However, the work proposes the use of subgoals and data-driven backward chaining rules. It has deductive capabilities, and detects satisfied conditions in business rules (using backward chaining), as soon as relevant facts become available. In our work, we focus rather on complex event detection, and enable a framework for event processing in pure Logic Programming style [18]. Our framework can accommodate not only events but conditions and actions (i.e. reactions on events), too. As this is not a topic of this paper, an interested reader is referred to our previous work [8].

Furthermore, IBM has been developing an event processing tool

[21]If a request is represented as a query (what is a usual case).

[22]CLIPS: `http://clipsrules.sourceforge.net/`

[23]TIBCO BusinessEvents: `http://www.tibco.com/software/complex-event-processing/businessevents/businessevents.jsp`

[24]Jess: http://jessrules.com/

[25]Drools: `http://jboss.org/drools/`

[26]BizTalk Rules Engine: `http://msdn.microsoft.com/en-us/library/dd879260\%28BTS.10\%29.aspx`

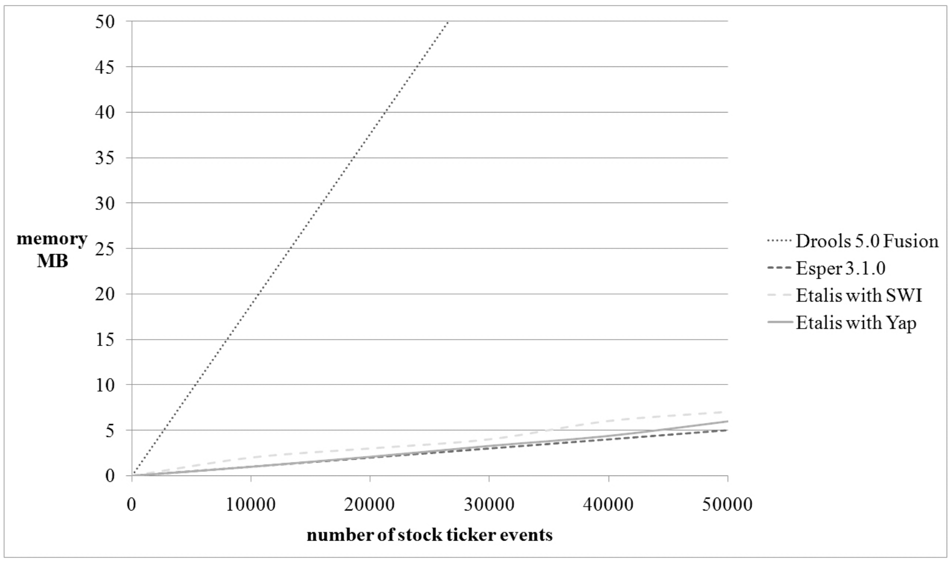

Figure 4: The memory consumption for the Stock Ticker experiment

available in IBM Tivoli Enterprise Console[27]. The engine is capable of processing event patterns and executing ECA rules. The approach uses the Prolog programming language to implement a formalism for specifying event patterns, event filters, and actions. While the formalism itself is very expressive, the whole approach is rather complex and hence somewhat unwieldy. We aim at providing a powerful, yet simple and intuitive declarative framework using Prolog for event and action processing.

Concluding this section, many mentioned studies aim to use more formal semantics in event processing. Our approach based on ETALIS Language for Events may also be seen as an attempt towards that goal. It features data-driven computation of complex events as well as rich deductive capabilities.

9. CONCLUSIONS AND FUTURE WORK

We propose a novel approach for Complex Event Processing based on the ETALIS Language for Events. The language comes with a clear declarative, formal semantics for complex event patterns. Further, our contribution includes an execution model which detects complex events in a data-driven fashion (based on *goal-directed event-driven rules*). We have also implemented a prototype of our approach, which allows for specification of complex events and their detection at occurrence time. The approach clearly substantiates existing event-driven systems with declarative semantics, extends capabilities of existing event-driven systems with the power of rule-based reasoning. We believe that the proposed approach is also more pragmatic from the implementation and optimization point of view (as many optimizations from Logic Programming and deductive databases are also applicable in our framework). Our experimental results, presented here, are encouraging.

For the next steps we will continue to study advantages and drawbacks of implementing Complex Event Processing in a logical framework, and continue our work on the implementation of new event operations. Finally, our intention is to extend the proposed formalism for event-driven Adaptable Business Processes. Adaptation in these business processes is driven by complex events at run-time, and the main challenge is to enable on-line verification of changes caused by events. Therefore our logical framework with strong inference capabilities can be seen as viable solution for this problem.

[27]IBM Tivoli Enterprise Console: `http://www-01.ibm.com/software/tivoli/products/enterprise-console/`

10. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] R. Adaikkalavan and S. Chakravarthy. Snoopib: Interval-based event specification and detection for active databases. In *Data Knowledge Engineering*. Elsevier Science Publishers B. V., 2006.

[3] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *SIGMOD*, 2008.

[4] J. F. Allen. Maintaining knowledge about temporal intervals. In *Communications of the ACM 26, 11, 832-843*, 1983.

[5] D. Anicic, P. Fodor, R. Stühmer, and N. Stojanovic. Event-driven transaction logic inference system (etalis)-implementation. `http://code.google.com/p/etalis/`.

[6] D. Anicic, P. Fodor, R. Stühmer, and N. Stojanovic. An approach for data-driven and logic-based complex event processing, abstract paper. In *DEBS*, 2009.

[7] D. Anicic, P. Fodor, R. Stühmer, and N. Stojanovic. Event-driven approach for logic-based complex event processing. In *CSE*, 2009.

[8] D. Anicic and N. Stojanovic. Expressive logical framework for reasoning about complex events and situations. In *Intelligent Event Processing - AAAI Spring Symposium 2009*. Stanford University, California, 2009.

[9] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs. In *PODS '86, Massachusetts, United States*. ACM, 1986.

[10] F. Bry and M. Eckert. Rule-based composite event queries: The language xchangeeq and its semantics. In *RR*. Springer, 2007.

[11] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S. Kim.

Composite events for active databases: Semantics, contexts and detection. In *VLDB*, 1994.

[12] S. Chakravarthy and D. Mishra. Snoop: an expressive event specification language for active databases. In *Data Knowledge Engineering*. Elsevier Science Publishers B. V., 1994.

[13] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. In *Journal of the ACM*. ACM, 1996.

[14] C. L. Forgy. Rete: a fast algorithm for the many pattern/many object pattern match problem. In *Artificial Intelligence*, 1982.

[15] A. Gupta and I. S. Mumick. Magic sets and other strange ways to implement logic programs. In *IEEE Data Engineering Bulletin*, 1985.

[16] P. Haley. Data-driven backward chaining. In *International Joint Conferences on Artificial Intelligence*. Milan, Italy, 1987.

[17] G. Lausen, B. Ludäscher, and W. May. On active deductive databases: The statelog approach. In *ILPS'97*, 1998.

[18] J. W. Lloyd. *Foundations of Logic Programming*. Computer Science Press, 1989.

[19] I. Motakis and C. Zaniolo. Composite temporal events in active database rules: A logic-oriented approach. In *Deductive and Object-Oriented Databases*. Springer-Verlag, 1995.

[20] A. Paschke, A. Kozlenkov, and H. Boley. A homogenous reaction rules language for complex event processing. In *International Workshop on Event Drive Architecture for Complex Event Process*. ACM, 2007.

[21] N. W. Paton and O. Díaz. Active database systems. In *ACM Comput. Surv.* ACM, 1999.

[22] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I*. W. H. Freeman & Co., New York, NY, USA, 2nd edition, 1990.

[23] E. Yoneki and J. Bacon. Unified semantics for event correlation over time and space in hybrid network environments. In *CoopIS/DOA/ODBASE 2005*. Springer-Verlag Berlin Heidelberg, 2005.